

Identifying Source Code File Experts

Otávio Cury
Federal University of Piauí
Teresina, Brazil
otaviocury@ufpi.edu.br

Guilherme Avelino
Federal University of Piauí
Teresina, Brazil
gaa@ufpi.edu.br

Pedro Santos Neto
Federal University of Piauí
Teresina, Brazil
pasn@ufpi.edu.br

Ricardo Britto
Blekinge Institute of Technology
Karlskrona, Sweden
rbr@bth.se

Marco Túlio Valente
Federal University of Minas Gerais
Belo Horizonte, Brazil
mtov@dcc.ufmg.br

ABSTRACT

Background: In software development, the identification of source code file experts is an important task. Identifying these experts helps to improve software maintenance and evolution activities, such as developing new features, code reviews, and bug fixes. Although some studies have proposed repository-mining techniques to automatically identify source code experts, there are still gaps in this area that can be explored. For example, investigating new variables related to source code knowledge and applying machine learning aiming to improve the performance of techniques to identify source code experts. **Aim:** The goal of this study is to investigate opportunities to improve the performance of existing techniques to recommend source code files experts. **Method:** We built an oracle by collecting data from the development history and surveying developers of 113 software projects. Then, we use this oracle to: (i) analyze the correlation between measures extracted from the development history and the developers' source code knowledge and (ii) investigate the use of machine learning classifiers by evaluating their performance in identifying source code files experts. **Results:** *First Authorship* and *Recency of Modification* are the variables with the highest positive and negative correlations with source code knowledge, respectively. Machine learning classifiers outperformed the linear techniques (F-Measure = 71% to 73%) in the public dataset, but this advantage is not clear in the private dataset, with F-Measure ranging from 55% to 68% for the linear techniques and 58% to 67% for ML techniques. **Conclusion:** Overall, the linear techniques and the machine learning classifiers achieved similar performance, particularly if we analyze F-Measure. However, machine learning classifiers usually get higher precision while linear techniques obtained the highest recall values. Therefore, the choice of the best technique depends on the user's tolerance to false positives and false negatives.

CCS CONCEPTS

• **Software and its engineering** → *Maintaining software.*

KEYWORDS

software maintenance, software evolution, mining software repository, source-code expertise, machine learning

ACM Reference Format:

Otávio Cury, Guilherme Avelino, Pedro Santos Neto, Ricardo Britto, and Marco Túlio Valente. 2022. Identifying Source Code File Experts. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '22)*, September 19–23, 2022, Helsinki, Finland. ACM, Helsinki, HEL, Finland, 12 pages. <https://doi.org/10.1145/3544902.3546243>

1 INTRODUCTION

Source code changes are fundamental activities during software evolution [58]. These changes are made in many development-related activities. Such activities require efficient management of the development team. However, this management becomes particularly complicated in large and geographically distributed projects, where project managers need as much information as possible about their development team to coordinate the project activities [30]. In this context, **knowing who has expertise in which parts of the source code is a very useful information**, especially in a context where remote work is growing fast and face-to-face interactions have been reduced [59].

Information on developers' expertise is valuable in various scenarios in software development. For example, it can be used in tasks assignment, such as to identify which experienced developer can help newcomers in implementing changes [38] or who is most suitable for bug fixing [2]. Additionally, this information helps to identify the concentration of knowledge in parts of the code [4, 21], i.e., situation that poses high risks to the future of the project.

However, due to the large amount of change-related information that developers and managers deal with every day [24], it is challenging to keep track of who is familiar with each project file. To help with this task, it is possible to rely on information available in Version Control Systems (VCS), wherein a large part of the developer-file iterations are logged. By using such information, several techniques were developed to automate the identification of experts in source code files [18, 24, 46, 48, 49].

Some research has been conducted to address the file expert identification problem. For example, in the work [7], the authors compared the performance of three techniques for identifying file

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEM '22, September 19–23, 2022, Helsinki, Finland

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9427-7/22/09...\$15.00

<https://doi.org/10.1145/3544902.3546243>

experts. They identified **an opportunity for improving the performance of existing techniques by adding information on file size and recency of modifications**. In this paper, we explore this opportunity by first analyzing the correlation between twelve measures extracted from the development history and the developers' source code knowledge. Following, we investigate the use of machine learning classifiers by evaluating their performance in identifying source code file experts on a large dataset composed of public and industrial software systems (including two projects from Ericsson). Particularly, we seek to answer the following research questions:

- **(RQ1)** *How do repository-based metrics correlate with developer's knowledge?*

Motivation: There are several works in the literature that use different repository-based metrics to infer the knowledge of developers in source code files. However, we did not identify studies that correlated these variables with knowledge. By answering this question, we seek to understand how these variables are related to knowledge in source code, which can guide the creation of models that estimate knowledge and help to identify source code experts.

- **(RQ2)** *How do machine learning classifiers compare with traditional techniques for identifying source code experts?*

Motivation: Due to the vastly successful application of machine learning classifiers in the software engineering literature, we believe that the application of machine learning classifiers can improve the performance in identifying experts achieved by other techniques in previous works.

The main contributions of this paper are twofold:

- (1) A correlation analysis between variables extracted from version control systems and developers' source code knowledge.
- (2) A comparative study on the performances of machine learning classifiers and three well-known techniques for identifying source code experts.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3 describes the procedure adopted to select the target subjects of the study, the compared techniques, and how we evaluate their performance. Sections 4 and 5 present the results of the comparison of the techniques and discuss the results, respectively. Section 6 lists threats to the validity of our results. Finally, Section 7 concludes by presenting our key findings.

2 RELATED WORK

We identified two main goals on research related to the identification of code experts: propose new techniques for the identification of source code experts and compare existing techniques. This section covers both types of works. Section 2.1 presents works that propose techniques to infer developers expertise on source code artifacts and Section 2.2 describes works that compare existing techniques.

2.1 Research that Proposes New Techniques

MacDonald and Ackerman [46] use a heuristic called *Line 10 rule* that prioritizes the developer who last changed a module in solving problems. Following the same premise, Hossen et al. [33] presented an approach called *iMacPro* that identifies experts associated with a change request based on who last changed certain files. Other works count the number of changes made on source code elements [9, 12, 28, 29, 49]. There are also studies that use information from files present in development branches to identify experts who perform merge operations involving these files [16, 17]. Other models, such as the one proposed by Sülün, Tüzün and Dogrusöz [64], use the number of commits in the artifact of interest and in related artifacts for the calculation of knowledge, in order to recommend code reviewers. In summary, these studies are based mainly on information about changes such as the number of commits and who made the last change to identify expertise. However, based on past works [7, 41], we suspect that these variables alone are not enough. For this reason, in this study, we analyze more variables and their relationship with developers' knowledge.

Other studies try to model the knowledge flow in the history of the source code. The *Degree of Knowledge* (DOK) model proposed by Fritz et al. [24] uses the information related to the *degree of authorship* (DOA) that the developer has with the code artifact, and the number of interactions (selections and edits) that the developer had with the artifact, named the *degree of interest* (DOI). However, the calculation of the DOI requires the use of special plugins in the development environment, which makes its usage impractical in a large study as the one we present in this paper. Regarding the differences for the models studied in this work, DOK does not deal with recency directly and does not consider the size of the file when estimating knowledge. These two variables were pointed out as important factors in the calculation of knowledge in previous works [7, 41].

Other techniques model the impact of time on the knowledge that developers have with source code artifacts. Silva et al. [18] presented a model that computes the developer's expertise in an entire (atomic) artifact, and also in its subparts (internal classes and methods), based on the number of changes made by a developer. The expertise analysis can be done using time windows that divide the history of an artifact into subsets of commits. Other approaches that consider the recency of changes appear in studies focused on the recommendation of developers for the resolution of change requests. Kagdi et al. [37] proposed an approach that locates source code files relevant for a given change request and identifies experts in those files using the *xFinder* [38, 39] approach, which prioritizes developers who made most commits in a given file. Tüzün and Dogrusöz [65], extend a previous work [64], by adding information on modification recency for the calculation of knowledge, aiming to recommend code reviewers. On one hand, these studies consider some measure of recency for identifying experts in source code files. On the other hand, they did not present an in-depth and large analysis that shows how the variables used are suitable for this identification.

In comparison to the data source used to extract knowledge information in source code, in this work we focus only on data contained in version control systems. Some works use other sources

such as: number of interactions with a file [23], code reviews [35, 68], numbers of meeting related to commits [35]. While these are valid data sources, they depend on specific tools, such as plugins installed in the development environment, the use of company-specific tools, and development culture. Due to the universality of version control systems in current software development [74], its use as a data source becomes easier in practice.

Regarding the use of machine learning, Montandon and colleagues [50] investigated the performance of supervised and unsupervised classifiers in identifying experts in three open-source libraries. Even though we followed a similar process for data collection and analysis, our work has a distinct purpose. We rely on classifiers for identifying experts at the level of source code files, while Montandon target the identification of experts in the use of libraries and frameworks, therefore using different variables than the ones used in this work. Other examples of machine learning applications in the context of developer expertise target the bug assignment problem [62], which is also a distinct problem than the one investigated here. In summary, **we have not identified studies that investigate the performance of machine learning in the classification of file experts based on VCS information**, such as our key goal in this work.

2.2 Comparison of Existing Techniques

Krüger and colleagues [41] analyzed the impact of forgetfulness of the developer about the code, using data from ten open-source repositories. They studied whether the forgetting curve described by Ebbinghaus [20] can be applied in the context of software development, and which variables influence the developer's familiarity with source code. They analyzed variables such as number of commits, changes made by other developers, percentage of code written by a developer in the current version of the file, and the behavior of tracking changes made by other developers.

Other works used techniques and models to identify expertise. Avelino and colleagues compared the performance of *Commits*, *Blame*, and *Degree-of-Authorship* (DOA) techniques in identifying source code file maintainers [7]. A survey similar to the one presented in this paper was made to create a dataset with data from eight open-source repositories and two private ones. The results showed that all three techniques have similar performance in identifying source code maintainers. However, the results also pointed out the importance of considering the recency of the modifications and the file size as a possible strategy to improve these techniques.

There are also papers that studied other types of expertise. For example, Hannebauer et al. [26] compared the performance of eight algorithms to recommend code reviewers. Out of these algorithms, six are based on expertise by modification and two are based on review expertise. The six algorithms based on expertise by modification are *Line 10 Rule* [46], *Number of Changes*, *Expertise Recommender* [46], *Code Ownership* [25], *Expertise Cloud* [1], and DOA. The algorithms based on review expertise are *File Path Similarity* (FPS) [67], and a model proposed by the authors, called *Weighted Review Count* (WRC). They used data from four FLOSS projects: Firefox, AOSP, OpenStack, and Qt. The algorithms based on review expertise performed better than the ones based on modification expertise, and the WRC algorithm achieved the best results.

Other works use information on expertise for the resolution of bug reports. Anvik and Murphy [3] compared two approaches to determine appropriate developers for resolving bug reports. One approach uses data from code repositories to define which developers are experts in the files associated with the bug report using the *Line 10* heuristic. The other approach uses data from bug networks such as the carbon-copy list (cc:), comments, and information from who resolved previous bugs. Development data from the Eclipse¹ platform was used. The authors concluded that the best approach depends on what users are looking for regarding precision and recall.

The work presented in this paper can be distinguished from the works described before in three key aspects: purpose, compared techniques, and scope. In relation to purpose, two works have the same objectives as ours: Avelino et al. [7] and Anvik and Murphy [3] (but they do not use the same techniques, particularly machine learning classifiers).

Regarding the compared techniques, Avelino et al. [7], Krüger et al. [41], and Hannebauer et al. [26] used the baseline techniques selected in this work. However, we also investigate the performance of machine learning models. Finally, regarding the scope of the studies, none of them used data from a similar number of repositories as in our study.

3 RESEARCH DESIGN

To achieve the objectives defined in this study, it is required a ground truth with data on source code experts. We build this ground truth by extracting data from open source and industrial projects. This ground truth is composed of the developers' knowledge in source code files and variables (or metrics) computed from the projects' development history.

3.1 Target Subjects

We selected open source repositories from the GitHub platform². To select these repositories, we adopted a similar procedure to other studies that investigate GitHub data [4, 60, 72]. First, for each of the six most popular programming languages on GitHub (Java, Python, Ruby, JavaScript, PHP, and C++)³ we selected the 50 most popular repositories as indicated by their number of stars. This measure is widely used by researchers in the selection of GitHub repositories [4, 13, 31, 36, 45, 52, 55, 60, 61], and perceived by developers as a reliable proxy of popularity [10]. Then, after cloning the repositories, we performed a repository filtering step based on three metrics: number of commits, number of files, and number of developers. As was done in a previous work [4], for each language, we removed repositories in the first quartile of the distribution of each metric, resulting in the intersection of the remaining sets. In other words, repositories with few commits, files, and developers were removed. Table 1 shows the first quartile of each of the three metrics for each programming language.

Additionally, we discarded repositories whose development history suggests that most of the software was developed outside of

¹<https://www.eclipse.org/eclipse/>

²<https://github.com/>

³The six most popular programming languages in 2019 <https://octoverse.github.com/#top-languages>

Table 1: First quartiles of filtering metrics, for each language

Language	Commits	Files	Developers
Python	510.75	87.50	45.25
Java	829.25	318.00	39.25
PHP	823.50	89.00	97.50
Ruby	1,650.25	152.75	198.25
C++	2,010.25	706.00	113.50
JavaScript	1,455.75	113.25	129.25

Table 2: Target open source repositories.

Language	Repos	Devs	Commits	Files
Python	25	18,936	192,587	39,154
Java	17	5,733	138,473	62,429
PHP	16	9,802	144,092	29,902
Ruby	25	38,036	605,546	88,869
C++	14	11,467	350,345	72,991
JavaScript	14	10,319	109,541	21,477
Total:	111	94,293	1,540,584	314,822

GitHub by removing repositories where more than half of its files were added in a few commits. As few commits, we considered the outliers of the distribution of the number of files added in each commit of a repository; we discarded repositories if most of their files (>50%) were added by the set of outliers commits, following the procedure done by Avelino and others [4].

The resulting dataset is composed of **111 popular GitHub repositories** distributed over the six most popular programming languages, which have a relevant number of developers, files, and commits. Table 2 summarizes the characteristics of these 111 repositories.

We also used data from **two industrial projects from Ericsson**⁴. Both projects were developed in the Java programming language. The development history of Project #1 has 74,078 commits, 17,329 files, and 513 contributors. On the other hand, Project #2 has 26,678 commits, 15,930 files, and 262 contributors. Therefore, the two industrial projects are also relevant according to the criteria applied to select the open-source repositories.

3.2 Ground Truth Construction

Extracting Development History: after downloading the selected repositories, we started the process of extracting their development history data. This data was extracted by collecting the commits from the *master/default-branch* of each repository, as described below.

First, we ran `git log --no-merge --find-renames` command to extract data from the commits logs of each repository. This command returns all commits that have no more than one parent (no-merge) and it automatically handles possible file renames.⁵ From each commit, three pieces of information were extracted: (1) changed files; (2) name and email of the commit’s author (developer who

performed the change); (3) type of the change: *addition*, *modification*, or *rename*.

After that, we discarded files that do not contain source code (e.g., images and documentation), third-party libraries, and files that are not part of one of the six programming languages considered in the study. To identify and discard these files we rely on the Linguist tool⁶.

Finally, we handled developer aliases by following the same procedure adopted in other works [4–7]. Aliases arise when a developer is associated with more than one pair (*name-dev*, *email*) on Git. For the purposes of this work, it is important to unify these contributors. This unification was performed in two stages. First, users who shared the same email, but with different names, were unified. Additionally, users with different emails, but with similar names were grouped. This similarity was computed using the *Levenshtein* distance [51], using a maximum modification threshold of 30% on the number of letters.

Generating Survey Sample: We used information extracted from development histories to create sample pairs (*developer*, *file*) for each repository. These samples are necessary to elaborate the survey applied in this study. They were created by performing the following steps for each repository:

- (1) We randomly selected a file and retrieved the list of developers who touched (created or modified) it.
- (2) We discarded the file if at least one of these developers reached the maximal limit of files we plan to send to a developer (*file_limit*), asking about his/her expertise on the file. Otherwise, we added the file to the list of each developer.
- (3) We repeated steps 1-2 until there are no more files to be verified.

By discarding files in each one of the developers already reached her *file_limit*, step 2 warrants that a file will be added to the sample only if it is possible to add all developers who modified it. This step aims to maximize the possibility of obtaining answers from all developers who touched a file. We established a *file limit* of five, seeking to not discourage developers from responding to the survey, which can happen according to guidelines in the literature [40, 43].

At the end of this step, a list of pairs (*developer*, *file*) was created for each repository. For open-source repositories, 20,564 pairs were generated, with 7,803 developers, 2.64 files per developer on average. For the two private repositories, 394 pairs were generated, with 92 developers, and 4.34 files per developer.

Sending the Survey: We conducted the survey by sending individual e-mails to each developer on the generated sample. The developers were invited to evaluate their knowledge in each of the files on their list, by using a scale from 1 (one) to 5 (five), where (1) means you have no knowledge about the file’s code; (3) means you would need to perform some investigations before reproducing the code; and (5) means you are an expert on this code

For the open-source repositories, 7,803 emails were sent, and 501 responses were received, representing a response rate of 7%. For the private repositories, 92 emails were sent, 38 responses were

⁴<https://www.ericsson.com/en>

⁵<https://git-scm.com/docs/git-log/1.5.6>

⁶<https://github.com/github/linguist/blob/master/lib/linguist/languages.yml>

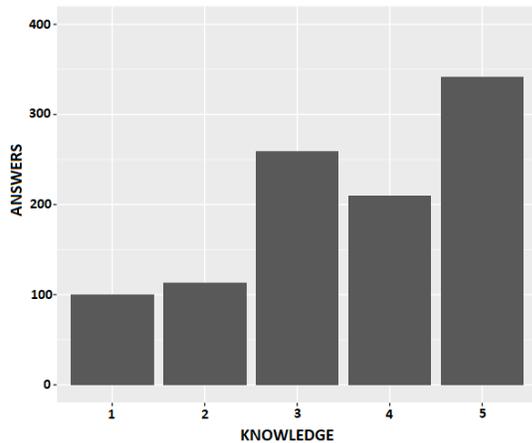


Figure 1: Responses distribution in the Public dataset.

received, representing a rate of 41%. From these responses, two datasets were created: one with 1,024 developer-file pairs coming from the answers of the open-source repositories, and a second dataset with 163 pairs extracted from the answers of the two private repositories. In the remainder of this paper, these two datasets will be named public and private datasets.

Processing the Answers: We process the answers by classifying each pair (*developer, file*) into one of two disjoint sets: *declared experts* (O_m), and *declared non-experts* ($O_{\bar{m}}$). A *declared expert* is a developer who claims to have knowledge of more than 3 (three) in a file; otherwise, he/she is a *declared non-expert*.

Figure 1 and 2 shows the distribution of responses in the public and private datasets respectively. **The public dataset is composed of 54% of declared experts and 46% of declared non-experts, and the private dataset is composed of 47% of declared experts and 53% of declared non-experts.** As we can see, the proportion of declared expert and declared non-expert sets are close in both datasets, and this class balance is an important property in classifier training [69].

3.3 Development Variables

Extracting Variables: We selected 12 variables (or features) that can be extracted from the development history. These variables and their meanings are shown in Table 3. Subsets of these variables are explored in other studies. Table 4 lists studies that analyzed these variables in the context of inferring source code knowledge. As shown, *NumCommits* is the most explored variable.

The variables *Adds*, *Dels*, *Mods*, and *Conds* are extracted using the command `git diff`⁷. This command returns the lines added and removed between two versions of a file. In this work, a modification is defined as a set of removed lines followed by a set of added lines of the same size [34, 44]. We use *Levenshtein* distance [51] to identify which pairs (*remove, add*) are modifications between two versions of the files. The algorithm takes the removed and added line as input, and returns a value that represents the number of

⁷<https://git-scm.com/docs/git-diff>

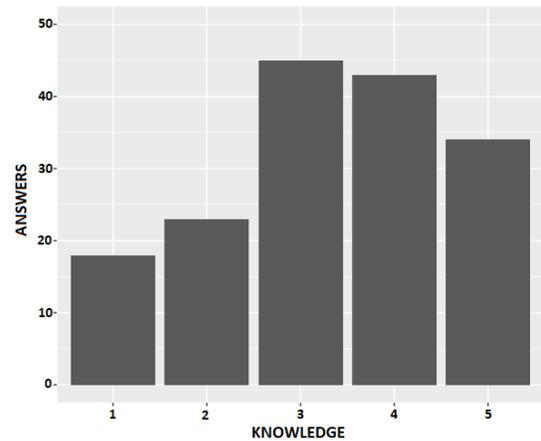


Figure 2: Responses distribution in the Private dataset

characters that must be modified to transform the removed line into the added line. In this work, a change is a modification if the returned value is less than a certain percentage (threshold) of the size of the removed line. A threshold of 40% was used, as suggested by Canfora et al. [11].

3.4 Compared techniques

This section describes the techniques used in this study to identify code experts.

Number of Commits: This technique counts the number of commits as a measure of knowledge that a developer has in a given source code file. The idea behind this technique is that a developer gains knowledge in a file by creating or modifying it. In other words, more commits are interpreted as more knowledge. This technique is widely used in the context of developer expertise, as shown in Table 4. It is applied individually [7, 18, 26], as well as combined with other techniques [24, 37, 66]. In this study, we use the number of commits individually as an expert identification technique. In this paper, we refer to this technique as *NumCommits*.

Blame: This technique infers the knowledge that a developer has in a file by counting the number of lines added by him and present in the last version of the file. Blame-like tools such as `git-blame`⁸ command are used to identify the authors of each file line. As in other works [7, 25, 26, 41, 57], we use the percentage of lines associated with a developer as a measure of knowledge.

Degree of Authorship (DOA): Fritz et al. [24] proposed that the knowledge of a developer on a source code file depends on factors such as the file's authorship, his/her number of contributions, and number of changes made by other developers. The authors combined these variables into a linear model called *Degree of Authorship* (DOA). The weights associated with each variable in this linear model were defined through an empirical study based on

⁸<https://git-scm.com/docs/git-blame>

Table 3: Variables extracted from the development history. The variables description are given considering a developer d and a file f in its last version.

Variable	Meaning
Adds	Number of lines added by a developer d on a file f
Dels	Number of lines deleted by a developer d on a file f
Mods	Number of lines modified by a developer d on a file f
Conds	Number of conditional statements added by a developer d on a file f
Amount	Sum of number of added and deleted lines of a developer d on a file
FA	Binary variable that indicates whether a developer d added the file f to the repository
Blame	Number of lines authored by a developer d that are in a file f
NumCommits	Number of commits made by a developer d on a file f
NumDays	Number of days since the last commit of a developer d on a file f
NumModDevs	Number of developers that committed on the file f since the last commit of a developer d
Size	Number of lines of code (LOC) of the file f
AvgDaysCommits	Average number of days between the commits of a developer d on a file f

Table 4: Extracted variables and related studies

Variable	Also used in these studies
Adds, Dels	[19, 25, 28, 34, 44, 49]
Mods, Conds, Amount	[19, 34, 44]
Blame	[7, 25, 26, 41, 57]
FA	[23, 24]
NumCommits	[1, 7, 15, 19, 24, 34, 41, 44, 46, 49, 50, 53] [18, 26, 37, 39, 48, 66]
NumDays	[7, 19, 34, 41, 44, 66]
NumModDevs	[19, 24, 34, 44]
Size	[7, 53]
AvgDaysCommits	[50]

data from the development of two systems. The DOA value that a developer d has in the version v of a f file is calculated by the following equation:

$$\text{DOA}(d, f(v)) = 3.293 + 1.098 * FA + 0.164 * DL - 0.321 * \ln(1 + AC) \quad (1)$$

where,

- FA : 1 if developer d is the creator of the file f , 0 otherwise.
- DL : is the number of changes made by developer d until version v of file f .
- AC : is the number of changes made by other developers in the file f up to version v .

Machine Learning Classifiers: We also investigate the applicability of machine learning algorithms for identifying source code experts. This proposal comes down to a binary classification. From the dataset described in Section 3.2, we labeled each pair (*developer*, *file*) in non-expert (knowledge 1-3, as their answers in the survey) and experts (knowledge 4-5). We choose the development variables (Table 4) *Adds*, *FA*, *Size*, and *NumDays* as features to train the machine learning models. The rationale for these choices is given in

Section 4, where we present the results of the *RQ1*. According to the values of these features, machine learning models can classify a developer as an expert or not from a source code file.

Five well-known machine learning classifiers are evaluated: Random Forest [42], Support-Vector Machines (SVM) [70], K-Nearest Neighbor (KNN) [56], Gradient Boosting Machine [22], and Logistic Regression [32]. We use *10-fold Cross-Validation* to evaluate the performance of the machine learning classifiers. 10-fold Cross-Validation consists of partitioning the data set in ten complementary subsets, and the use of each of these subsets for model training and the rest for model validation. In the end, the fitness measures are combined to obtain a more accurate estimate of the model's performance [8]. We use *F-Measure* as a performance measure. To find the best settings for these classifiers, we perform a grid search [14] to adjust the hyper-parameters and find the best settings for each classifier.

3.5 Linear Techniques Evaluation

To evaluate *NumCommits*, *Blame*, and *DOA* in the identification of software experts, we adopt a similar procedure as the one followed in a previous related work [7]. First, we normalize the technique values. For this purpose, we set as 1 the expertise of a developer d in file f if d has the highest value for a given technique in f ; otherwise, we set a proportional value. For example, suppose that for a given file f developers $d1$, $d2$, and $d3$ have a *Blame* value of 10, 15, and 20. Their normalized values for blame regarding the file f are as follows: $expertise(d1, f) = 10/20 = 0.5$, $expertise(d2, f) = 15/20 = 0.75$, and $expertise(d3, f) = 20/20 = 1$. We apply this normalization process to all techniques.

After the normalization, a developer d is classified as an expert of a file f if its *expertise* (d, f) is higher or equal than a threshold k ; otherwise he/she is considered a *non-expert*. In the example of the previous paragraph, considering a threshold $k = 0.7$, developers $d2$ and $d3$ would be considered experts of the f file accordingly with

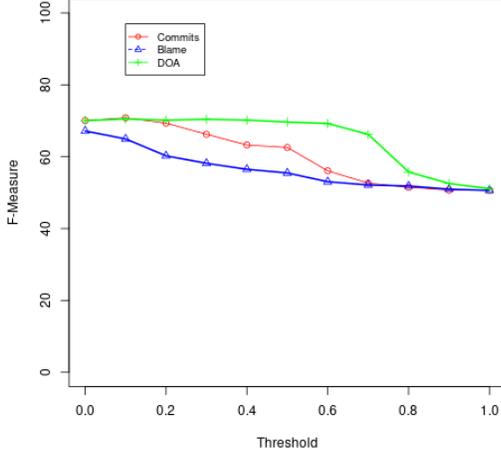


Figure 3: Performance at analyzed thresholds using public data

the *Blame* technique.

Evaluating Performance: We always compared the performance of the technique in their best settings. For the linear techniques, this requires setting the classification threshold k that maximizes the correct identification of file experts.

To this purpose, we first compute the performance of each technique by adopting 11 different thresholds, i.e., by varying the threshold k from 0 to 1, under 0.1 steps. At the end of this process, the best performance of the linear techniques is obtained together with their associated thresholds k . For each threshold, we use *10-fold Cross-Validation* to compute the *F-Measure* (F1-Score) of the techniques by applying the following equations:

$$Precision(k) = \begin{cases} \frac{|(d,f) \in O_m \mid expertise(d,f) > k|}{|expertise(d,f) > k|}, & \text{if } k = 0 \\ \frac{|(d,f) \in O_m \mid expertise(d,f) \geq k|}{|expertise(d,f) \geq k|}, & \text{otherwise} \end{cases} \quad (2)$$

$$Recall(k) = \begin{cases} \frac{|(d,f) \in O_m \mid expertise(d,f) > k|}{|O_m|}, & \text{if } k = 0 \\ \frac{|(d,f) \in O_m \mid expertise(d,f) \geq k|}{|O_m|}, & \text{otherwise} \end{cases} \quad (3)$$

$$F - Measure(k) = 2 * \frac{Precision(k) * Recall(k)}{Precision(k) + Recall(k)} \quad (4)$$

where O_m is the set of declared experts, as we inferred from the survey responses (Section 3.2).

Thresholds Calibration:

Figures 3 and 4 show the *F-Measure* results for each linear technique at each analyzed threshold, under the public and private datasets, respectively.

As we can see, *Blame* reaches its best performance when we use the lowest possible threshold ($k = 0$), in both public and private data. In other words, by adopting $k = 0$, *Blame* reaches its best

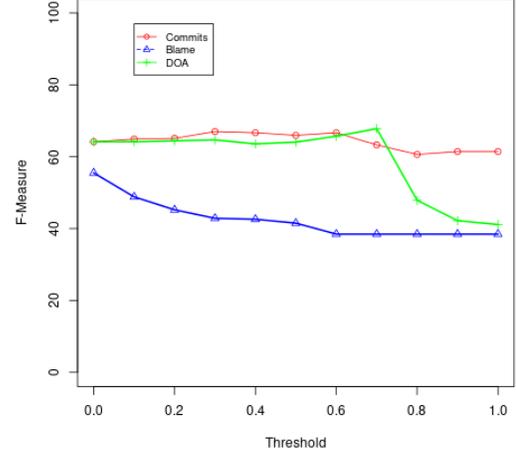


Figure 4: Performance at analyzed thresholds using private data

performance when it is used to classify as an expert any developer who has at least one line of code in the last version of the file. Similarly, *NumCommits* achieves the best performance by adopting the thresholds $k = 0.1$ and $k = 0.3$, in the public and private datasets, respectively. The low thresholds indicate that both techniques perform better by relying on minimal changes to consider developers as file experts. On the other hand, *DOA* has its best performance with a threshold $k = 0.1$ on public data; and $k = 0.7$, on private data.

4 RESULTS

This section presents the results of the two research questions proposed in this study. First, we present the correlation analysis between the extracted variables and developers' knowledge. Next, we present the performance of the techniques in our two datasets.

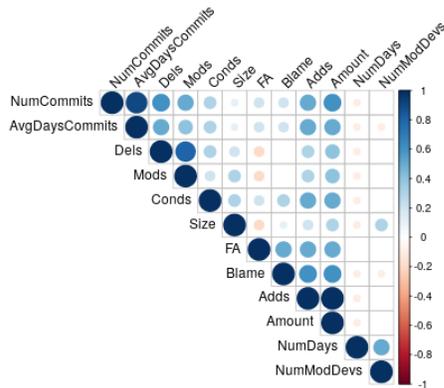
4.1 How do repository-based metrics correlate with developer's knowledge? (RQ1)

We analyze the correlations between the development variables and the developers' knowledge, as obtained in the survey. In the remainder of this paper, the survey responses will be represented by a variable named *Knowledge*. We answer *RQ1* and, consequently, identify which variables could be part of a prediction knowledge model.

Table 5 shows the directions and intensities of the correlations between the extracted variables and the *Knowledge* variable, by applying Spearman's *rho*, since it is suitable for data sets that do not follow the normal distribution. We consider results to be statistically significant when $p < 0.05$. Even though *NumCommits* is the most used variable for inferring file's knowledge in the literature, it does not show the strongest positive correlation with the *Knowledge* variable. The variable with the highest positive correlation is *FA*, closely followed by *Adds*. This suggests that a more fine-grained

Table 5: Correlation of extracted variables with Knowledge

Variable	Corr. with Knowledge	p-value
NumDays	- 0.24	<0.001
Size	- 0.21	<0.001
NumModDevs	- 0.21	<0.001
Mods	0.01	0.515
Dels	0.02	0.369
Cond	0.19	<0.001
NumCommits	0.20	<0.001
AvgDaysCommits	0.21	<0.001
Amount	0.28	<0.001
Blame	0.29	<0.001
Adds	0.31	<0.001
FA	0.33	<0.001

**Figure 5: Correlation between dependent variables**

measure of changes like *Adds* can be more important than *NumCommits* to compose a knowledge model. The variable with the lowest correlation with *Knowledge* is *Mods*. Finally, the variable that shows the highest negative correlation is *NumDays*, followed by *NumModDevs* and *Size*. These results reinforce the importance of recency for inferring the knowledge that a developer has about a source code file.

We also analyze the correlation between the extracted variables using Spearman’s *rho* [63]. These correlations are represented in Figure 5, where colors close to 1 and -1 represent higher positive and negative correlations, respectively. This analysis is relevant because variables with a certain level of correlation can lead to inaccurate models [73].

As expected, *NumCommits* has a strong correlation with *Adds*, *Dels*, *Mods*, *Amount*, and *AvgDaysCommits* ($\rho \geq 0.5$). Therefore, any one of these five variables can be used to measure the number of changes made by a developer throughout the history of a file.

The variables *NumModDevs* and *NumDays* also show a moderate positive correlation ($\rho \geq 0.5$) with each other.

Summary of RQ1: *First Authorship* (FA) has the highest positive correlation with knowledge in source code files and *Recency of Modification* has the highest negative correlation. Therefore, this result suggests that file creators tend to have high knowledge on the files they created; however, this knowledge decreases with time.

Variable Selection Rationale: In a previous research, Avelino and colleagues described how the variables *file size* and *recency of modification* influence developer’s knowledge on source code files [7]. Thus, models that take this information into account tend to be more accurate. This previous finding is reinforced by the results in Table 5, where these variables achieved the highest negative correlation with *Knowledge*. Therefore, we include the variables *Size* and *NumDays* in the proposed machine learning models. Considering the variables *Adds*, *Amount*, and *Blame*, we choose the variable *Adds*, as it has the highest positive correlation with *Knowledge* among the three. No more than one of the three variables is chosen because these variables are conceptually related, as commented in Section 3.3.

The variable *NumCommits* is not included in the models. It can be viewed as just a macro way of accounting for changes made by a developer to a source code file, and the variable *Adds* has already been chosen, since it has a higher positive correlation with *Knowledge*. In addition, *Adds* has a moderate correlation with *NumCommits* ($\rho \geq 0.5$, Figure 5). For a similar reason, *AvgDaysCommits* is not included in the model.

We also add the variable *FA*, which has the highest positive correlation with *Knowledge* among all variables. The variable *NumModDevs* is not included due to its moderate correlation with *NumDays* ($\rho \geq 0.5$).

4.2 How do machine learning classifiers compare with traditional techniques in identifying source code experts? (RQ2)

After identifying and selecting the variables with the highest correlations with source code knowledge, we train the machine learning models using them as features. After that, we compared the performance of these models with those of linear models in the identification of experts.

Table 6 presents the performance of linear techniques and machine learning classifiers, in the two analyzed scenarios. Regarding the public dataset (Table 6 - **Public**), *DOA* and *NumCommits* had the best F-Measure (70%), followed by *Blame*, with a F-Measure of 67%. Regarding recall, the technique with the best result was *DOA* (97%). The technique with the lowest recall in the same scenario is *Blame* (83%). Finally, the technique with the highest precision is *NumCommits* (60%), while *DOA* has the lowest precision (55%).

When using the private dataset (Table 6 - **Private**), *DOA* had the highest value for F-Measure (68%), closely followed by *NumCommits* (67%). As in the other scenario, *Blame* had the worst F-Measure (55%). The best Recall was from *NumCommits* (86%), with *Blame* presenting the lowest recall (62%). Regarding precision, *DOA* achieved the best result (61%), and *Blame* the lowest one (50%).

Table 6: Performance of linear and machine learning techniques

	Public			Private		
	Precision	Recall	F-Measure	Precision	Recall	F-Measure
DOA	0.55	0.97	0.70	0.61	0.77	0.68
NumCommits	0.60	0.87	0.70	0.55	0.86	0.67
Blame	0.56	0.83	0.67	0.50	0.62	0.55
Random Forest	0.73	0.74	0.73	0.59	0.62	0.60
SVM	0.71	0.75	0.73	0.63	0.60	0.61
KNN	0.71	0.71	0.71	0.70	0.69	0.67
GBM	0.73	0.73	0.73	0.65	0.58	0.60
Logistic Regression	0.69	0.75	0.72	0.70	0.51	0.58

When we consider the scenario using the public dataset (Table 6 - **Public**), *Random Forest*, *SVM*, and *GBM* had the best F-Measure (73%), with the other two classifiers reaching close values. The classifiers with the highest recall were *SVM* and *Logistic Regression* (75%), and *KNN* obtained the lowest result (71%). In terms of precision, *GBM* and *Random Forest* have the highest values (73%). The lowest precision is from *Logistic Regression* (69%).

Regarding the private dataset (Table 6 - **Private**), the classifier with the highest F-Measure was *KNN* (67%) and *Logistic Regression* had the lowest one (58%). The best recall was also obtained by *KNN* (69%) and *Logistic Regression* had the lowest one (51%). Finally, the best precision was achieved by *Logistic Regression* and *KNN* (70%). *Random Forest* had the lowest precision (59%).

Summary of RQ2: In the public dataset, machine learning classifiers outperformed the linear techniques (*F-Measure* = 0.71 to 0.73). In the private dataset, this advantage is not clear, with *F-Measure* ranging from 0.55 to 0.68 for the linear techniques and 0.58 to 0.67 for ML techniques.

5 DISCUSSION

In this section, we start by highlighting our key findings:

- (1) The linear techniques (DOA, Commits, and Blame) tend to have low precision, but high recall in the two analyzed datasets (public and industrial projects). In other words, they tend to positively classify developers as experts, but at the cost of a significant number of false positives.
- (2) Regarding the linear techniques, DOA has the best performance, in both datasets.
- (3) The ML classifiers have a very distinct performance in the investigated datasets. However, we must highlight the size difference of both datasets: there are 1,024 datapoints in the public dataset and only 163 in the private one. This difference can have a major impact on algorithms that include a training phase. In other words, ML techniques tend to improve their performance as the size of the training set increases, up to a certain point of stability, a behavior called the *Learning Curve* [47]. For this reason, we argue the public dataset results are more representative of the performances of such techniques.
- (4) Considering the public dataset, the best ML classifiers (in terms of F-Measure) are *Random Forest*, *SVM*, and *Gradient*

Boosting Machines. On the other hand, in the private dataset, *K Nearest Neighbors* reached the best performance.

- (5) Even though there are linear techniques with similar *F-Measure*, the machine learning classifiers show a better balance between *Precision* and *Recall*.

However, overall, we acknowledge that the linear techniques and the machine learning classifiers achieved similar performance, particularly if we analyze F-Measure. For this reason, **the choice of the best technique depends on the user's tolerance to false positives and false negatives**. Particularly, we envision two practical application scenarios:

- When the application requires finding few or even just one expert, such as when performing a merge operation [16] or onboarding a new team member [39], machine learning classifiers are more recommended as they are more precise.
- When the application demands more experts, such as when we need to evolve or maintain a more complex feature [33], it is more plausible to tolerate some false positives. In such cases, linear techniques are recommended.

6 THREATS TO VALIDITY

This section describes the threats to the validity of this study based on four categories: construct, internal, conclusion, and external validity [71].

Construct Validity: Some of the developers who participated in the survey may have inflated their self-assessment knowledge, due to some fear regarding the commercial use of this information. To tackle this problem, we clarified in the survey that their answers would be used only for academic purposes. Regarding the risk of misinterpretation of the knowledge scale (1 to 5), we provided a guide to the scores meaning. However, even the concept of an expert is subjective, which remains a threat. Another threat is the definition of expert according to the survey responses. Developers with knowledge above three were considered file experts. We claim this is a more conservative division of the knowledge scale, already applied on a similar study [7].

Internal Validity: There are other factors that may have an influence on the knowledge that a developer has in a source code file, which might not have been taken into account in this study.

For example, knowledge can be acquired in activities that do not require commits in VCS. However, in this work we limit ourselves to identify experts using authorship information that can be extracted from version control systems, which are tools widely used in software development, making the evaluated techniques applicable to most projects. Nevertheless, there are other variables that can be extracted from the information contained in code repositories, which can also play an important role in the process of identifying file experts. Among them, we can mention the iteration with files in the same module and the complexity of the changed code. However, we perceive these variables as more complex, more difficult to compute, and dependent on the programming language and design of the target systems. For this reason, we focused on variables that are less dependent on the particularities of the software under analysis.

Conclusion Validity: Regarding the statistical tests, Spearman's ρ is used to analyze the correlation between the extracted variables. This coefficient was chosen because it does not require a normal distribution or a linear relationship between the variables. However, there is no wide consensus on the interpretation of the correlation values returned by the test. However, we relied on interpretations that are also used in studies from different areas [54, 63]. We use 10-fold cross-validation to assess the performance of the machine learning models. We emphasize the classifier results may vary to some degree according to the number of folds chosen in the cross-validation. However, in general, five or 10-folds are recommended as good parameters [27].

External Validity: Some decisions were taken to maximize the ability to generalize the study results. For open-source systems, we choose six popular programming languages. Therefore, by not limiting the projects to a single programming language, we intended to reduce the impact that certain languages may have on the developers' familiarity assessment, and consequently in our results. In total, data from 113 systems are used. The small amount of data from private projects makes it difficult to generalize the results for this type of project, but these results can be generalized to projects with characteristics similar to those analyzed.

7 CONCLUSION

In this paper, we investigated the use of predictive models for the identification of source code file experts based on information available in version control systems. From a large dataset composed of public and industrial projects, we identified the variables that are most related to source code knowledge (RQ1). *First Authorship* and *Number of Lines Added* shown highest positive correlations with knowledge, while *Recency* and *File Size* have the highest negative correlation. Using the best variables in this analysis, machine learning models were evaluated for the identification of experts (RQ2). The performance of these models was compared with other techniques previously studied in the literature. As a result, the ML algorithms achieved the best F-Measure and precision among the compared techniques, with highlights for *Random Forest*, *Support Vector Machines*, and *Gradient Boosting Machines*.

Our findings can support the design and implementation of tools that seek to automate the process of identifying file experts in

software projects. We also provided insights on the variables that are most relevant to the derivation of new knowledge models. The models evaluated in this paper can be part of future investigations and research in repository analysis. In addition, future research can assess the performance of the studied techniques in other industrial contexts.

As a final note, we provide a replication package with the results of our analysis as well as the survey's answers, repositories data, and scripts used in the paper. The replication package is available at <https://doi.org/10.5281/zenodo.6757349>.

8 COMPETING INTERESTS

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

ACKNOWLEDGMENTS

We would like to thank the 501 developers (Ericsson and Open Source projects) who took their time to answer the survey and provided essential information to our investigation. Additionally, we also thank CAPES, CNPq, FAPEMIG, and UFPI for supporting this work.

REFERENCES

- [1] Omar Alonso, Premkumar T Devanbu, and Michael Gertz. 2008. Expertise identification and visualization from CVS. In *Proceedings of the 2008 international working conference on Mining software repositories*. 125–128.
- [2] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *Proceedings of the 28th international conference on Software engineering*. 361–370.
- [3] John Anvik and Gail C Murphy. 2007. Determining implementation expertise from bug reports. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 2–2.
- [4] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. 2016. A novel approach for estimating truck factors. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.
- [5] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. 2017. Assessing code authorship: The case of the Linux kernel. In *IFIP International Conference on Open Source Systems*. Springer, Cham, 151–163.
- [6] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. 2019. Measuring and analyzing code authorship in 1 + 118 open source projects. *Science of Computer Programming* 176 (may 2019), 14–32.
- [7] Guilherme Avelino, Leonardo Passos, Fabio Petrillo, and Marco Tulio Valente. 2018. Who Can Maintain This Code?: Assessing the Effectiveness of Repository-Mining Techniques for Identifying Software Maintainers. *IEEE Software* 36, 6 (2018), 34–42.
- [8] Daniel Berrar. 2019. Cross-validation. *Encyclopedia of Bioinformatics and Computational Biology* 1 (2019), 542–545.
- [9] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code! Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 4–14.
- [10] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.
- [11] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2007. Identifying changed source code lines from version repositories. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 14–14.
- [12] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2012. Who is going to mentor newcomers in open source projects?. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.
- [13] Diego Castro and Marcelo Schots. 2018. Analysis of test log information through interactive visualizations. In *Proceedings of the 26th Conference on Program Comprehension*. 156–166.
- [14] Marc Claesen and Bart De Moor. 2015. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127* (2015).

- [15] Eleni Constantinou and Georgia M Kapitsaki. 2016. Identifying developers' expertise in social coding platforms. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 63–67.
- [16] Catarina Costa, Jair Figueiredo, Leonardo Murta, and Anita Sarma. 2016. TIP-Merge: recommending experts for integrating changes across branches. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 523–534.
- [17] Catarina de Souza Costa, Jose Jair Figueiredo, Joao Felipe Pimentel, Anita Sarma, and Leonardo Gresta Paulino Murta. 2019. Recommending Participants for Collaborative Merge Sessions. *IEEE Transactions on Software Engineering* (2019), 1–1. <https://doi.org/10.1109/TSE.2019.2917191>
- [18] Jose Ricardo da Silva, Esteban Clua, Leonardo Murta, and Anita Sarma. 2015. Niche vs. breadth: Calculating expertise over time through a fine-grained analysis. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 409–418.
- [19] Francisco Vanderson de Moura Alves, Pedro de Alcântara dos Santos Neto, Werney Ayala Luz Lira, and Irvayne Matheus de Sousa Ibiapina. 2018. Analysis of Code Familiarity in Module and Functionality Perspectives. In *Proceedings of the 17th Brazilian Symposium on Software Quality*. 41–50.
- [20] Hermann Ebbinghaus. 1885. *Über das Gedächtnis: untersuchungen zur experimentellen psychologie*. Duncker & Humblot.
- [21] Mivian Ferreira, Marco Tulio Valente, and Kecia Ferreira. 2017. A comparison of three algorithms for computing truck factors. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 207–217.
- [22] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.
- [23] Thomas Fritz, Gail C Murphy, and Emily Hill. 2007. Does a programmer's activity indicate knowledge of code?. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 341–350.
- [24] Thomas Fritz, Gail C Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. 2014. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 2 (2014), 1–42.
- [25] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. 2005. How developers drive software evolution. In *Eighth international workshop on principles of software evolution (IWPSE'05)*. IEEE, 113–122.
- [26] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. 2016. Automatically recommending code reviewers based on their expertise: An empirical comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 99–110.
- [27] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media.
- [28] Lile Hattori and Michele Lanza. 2009. Mining the history of synchronous changes to refine code ownership. In *2009 6th IEEE international working conference on mining software repositories*. IEEE, 141–150.
- [29] Lile Hattori and Michele Lanza. 2010. Syde: a tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 2*. 235–238.
- [30] James D Herbsleb and Rebecca E Grinter. 1999. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st international conference on Software engineering*. 85–95.
- [31] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 426–437.
- [32] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Vol. 398. John Wiley & Sons.
- [33] Md Kamal Hossen, Huzefa Kagdi, and Denys Poshyvanyk. 2014. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22nd International Conference on Program Comprehension*. 130–141.
- [34] Irvayne M. S. Ibiapina, F. V. M. Alves, Werney A. L. Lira, Gleison A. Silva, and Pedro A. S. Neto. 2017. Inferência da Familiaridade de Código por Meio da Mineração de Repositórios de Software. *Simpósio Brasileiro de Qualidade de Software - SBQS* (2017).
- [35] Elgun Jabrayilzade, Mikhail Evtkhiev, Eray Tüzün, and Vladimir Kovalenko. 2022. Bus Factor In Practice. *arXiv preprint arXiv:2202.01523* (2022).
- [36] Jing Jiang, David Lo, Xinyu Ma, Fuli Feng, and Li Zhang. 2017. Understanding inactive yet available assignees in GitHub. *Information and Software Technology* 91 (2017), 44–55.
- [37] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Maen Hammad. 2012. Assigning change requests to software developers. *Journal of software: Evolution and Process* 24, 1 (2012), 3–33.
- [38] Huzefa Kagdi, Maen Hammad, and Jonathan I Maletic. 2008. Who can help me with this source code change?. In *2008 IEEE International Conference on Software Maintenance*. IEEE, 157–166.
- [39] Huzefa Kagdi and Denys Poshyvanyk. 2009. Who can help me with this change request?. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 273–277.
- [40] Mark Kasunic. 2005. *Designing an effective survey*. Technical Report. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.
- [41] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do you remember this source code?. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 764–775.
- [42] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.
- [43] Johan Linaker, Sardar Muhammad Sulaman, Martin Höst, and Rafael Maiani. 1.1. *Lund University* (2015).
- [44] Werney Ayala Luz Lira. 2016. *Um método para inferência da familiaridade de código em projetos de software*. Master's thesis. Universidade Federal do Piauí, Teresina.
- [45] Davood Mazinianian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. 2017. Understanding the use of lambda expressions in Java. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–31.
- [46] David W McDonald and Mark S Ackerman. 2000. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. 231–240.
- [47] Christopher Meek, Bo Thiesson, and David Heckerman. 2002. The learning-curve sampling method applied to model-based clustering. *Journal of Machine Learning Research* 2, Feb (2002), 397–418.
- [48] Shawn Minto and Gail C Murphy. 2007. Recommending emergent teams. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 5–5.
- [49] Audris Mockus and James D Herbsleb. 2002. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 503–512.
- [50] Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Identifying experts in software libraries and frameworks among GitHub users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 276–287.
- [51] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM computing surveys (CSUR)* 33, 1 (2001), 31–88.
- [52] Sebastian Nielebock, Robert Heumüller, and Frank Ortmeier. 2019. Programmers do not favor lambda expressions for concurrent object-oriented code. *Empirical Software Engineering* 24, 1 (2019), 103–138.
- [53] Johnatan Oliveira, Markos Viggiano, and Eduardo Figueiredo. 2019. How Well Do You Know This Library? Mining Experts from Source Code Analysis. In *Proceedings of the XVIII Brazilian Symposium on Software Quality*. 49–58.
- [54] Brian R Overholser and Kevin M Sowinski. 2008. Biostatistics primer: part 2. *Nutrition in clinical practice* 23, 1 (2008), 76–84.
- [55] Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. 2014. A study of external community contribution to open-source projects on GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 332–335.
- [56] Leif E Peterson. 2009. K-nearest neighbor. *Scholarpedia* 4, 2 (2009), 1883.
- [57] Foyzur Rahman and Premkumar Devanbu. 2011. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*. 491–500.
- [58] Václav Rajlich. 2014. Software evolution and maintenance. In *Proceedings of the on Future of Software Engineering*. 133–144.
- [59] Paul Ralph, Sebastian Baltes, Gianisa Adisaputri, Richard Torkar, Vladimir Kovalenko, Marcos Kalinowski, Nicole Novielli, Shin Yoo, Xavier Devroey, Xin Tan, et al. 2005. Pandemic programming: how COVID-19 affects software developers and how their organizations can help (2020). *arXiv preprint arXiv:2005.01127* (2005).
- [60] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 155–165.
- [61] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseider, and Hanspeter Mössenböck. 2018. An analysis of x86-64 inline assembly in C programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 84–99.
- [62] Ali Sajedi-Badashian and Eleni Stroulia. 2020. Guidelines for evaluating bug-assignment research. *Journal of Software: Evolution and Process* (2020), e2250.
- [63] Patrick Schober, Christa Boer, and Lothar A Schwarte. 2018. Correlation coefficients: appropriate use and interpretation. *Anesthesia & Analgesia* 126, 5 (2018), 1763–1768.
- [64] Emre Sülün, Eray Tüzün, and Uğur Doğrusöz. 2019. Reviewer recommendation using software artifact traceability graphs. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. 66–75.
- [65] Emre Sülün, Eray Tüzün, and Uğur Doğrusöz. 2021. RSTrace+: Reviewer suggestion using software artifact traceability graphs. *Information and Software*

- Technology* 130 (2021), 106455.
- [66] Xiaobing Sun, Hui Yang, Xin Xia, and Bin Li. 2017. Enhancing developer recommendation with supplementary information via mining historical commits. *Journal of Systems and Software* 134 (2017), 355–368.
- [67] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. 2014. Improving code review effectiveness through reviewer recommendations. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. 119–122.
- [68] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*. 1039–1050.
- [69] Gary M Weiss and Foster Provost. 2001. The effect of class distribution on classifier learning: an empirical study. (2001).
- [70] Jason Weston and Chris Watkins. 1998. *Multi-class support vector machines*. Technical Report. Citeseer.
- [71] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.
- [72] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, Ahmed E Hassan, and Naoyasu Ubayashi. 2015. Revisiting the applicability of the pareto principle to core development teams in open source software projects. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*. 46–55.
- [73] Lei Yu and Huan Liu. 2003. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proceedings of the 20th international conference on machine learning (ICML-03)*. 856–863.
- [74] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. 2018. Version control system: A review. *Procedia Computer Science* 135 (2018), 408–415.